School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

UML 2.0 with VizzAnalyser

Francisco Modesto

Sep
2007

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 07121
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--07121/--SE

Bachelor Thesis

UML 2.0 with VizzAnalyzer

Francisco Modesto

September 12th, 2007

Department of Computer Science
School of Mathematics and Systems Engine
Växjö University

Supervisor:
Rüdiger Lincke, Phil. Lic

Abstract:
Analyzing software contains two different tasks. First of all we are analyzing the software and try to calculate some metrics for software quality. Then those results have to be presented to the software engineers. VizzAnalyzer is a tool for analysis and visualization of software. It visualization allow not for a standardized diagram representation. Therefore it is difficult for others to understand, and we need to explain the meaning of our non-standard diagram elements. The solution is to use a standardized representation which can be understood by both sides. UML is such a collection of intuitively diagrams with standardized elements. Their meaning is clear to most software engineers.

We extended our analysis tool, the VizzAnalyzer, allowing it to view software systems as UML Class diagrams. We reused the existing plug-in architecture to connect our analysis tool with yEd, a graph visualization program. This plug-in is responsible for exchanging the data between the two applications.

We solve this conversion defining an UML Class Diagram Model and the mapping function between this model and the Common Meta-Model used by VizzAnalyzer and our Class Diagram Model. After that, we export this Class Diagram Model to a format suitable for yEd to display.

Now we can generate Class Diagrams with the VizzAnalyzer tool. This will allow a better communication of the results derived by different analysis with the software engineers.

This thesis describes the evolution of different alternatives and the design and implementation of our solution.

Keywords: VizzAnalyzer, Common-Meta-Model, Class Diagram, UML Layout, Framework, Plug-in.

Content

1 INTRODUCTION....1
1.1 Context of the Thesis....1
1.2 Problem....2
1.3 Goals and Criteria....2
1.4 Motivation....2
1.5 Outline....3
2 BACKGROUND....4
2.1 VizzAnalyzer....4
2.2 Common Meta-Model....4
2.3 GRAIL....5
2.4 UML Class diagram....5
2.5 yEd....6
3 REQUIREMENTS....7
3.1 Users....7
3.2 Use-Cases....7
3.3 Functional Requirements....9
3.4 Non-Functional Requirements....9
4 ANALYSIS....11
4.1 Mapping from the CMM to UML Class Diagram Model....11
4.2 Visualization (of the Class Diagram)....12
4.2.1 Class diagram viewer....13
4.2.2 UML Layout....15
4.3 Discussion....17
5 DESIGN & IMPLEMENTATION....19
5.1 The VizzAnalyzer Framework....19
5.1.1 Architecture....19
5.1.2 GRAIL and the Common Meta-Model....20
5.1.3 The Visualization plug-in....20
5.2 Transformation process....20
5.3 Visualization process....22
5.3.1 VizzAnalyzer yEd plug-in....22
5.3.2 GML File Format....23
5.3.3 yEd extension of GML format....24
5.3.4 Export the file....25
6 CONCLUSION AND FUTURE WORK....28
6.1 Conclusions....28
6.2 Future Work....29
7 REFERENCES....30
APPENDIX A - USER MANUAL....31
A.1 System Requirements....31
A.2 Installation....31
A.3 UML plug-in....31
A.3.1 Loading a graph....31
A.3.2 Run the plug-in....31
A.4 yEd Result....32
A.4.1 Resize nodes....33
A.4.2 Apply layout....34
A.4.3 Final result....35
A.5 Uninstall....35

Table of Figures

Figure 2.1: Software Analysis and Visualization Process....4
Figure 2.2: Different representations of a class property....5
Figure 2.3: Different UML relationships....6
Figure 3.1: VizzAnalyzer Use-Case....7
Figure 3.2: Visualization Use-Case....8
Figure 4.1: Common Meta-Model productions P and specialization hierarchy....11
Figure 4.2: Common Meta-Model, binary semantic relations in R....11
Figure 4.3: UML Class Model productions P and specialization hierarchy....12
Figure 4.4: UML Class Model, binary semantic relations in R....12
Figure 4.5: Mapping to the UML model....12
Figure 4.6: UML Hierarchical Layout....16
Figure 4.7: UML Directed Orthogonal Layout....17
Figure 5.1: VizzAnalyzer Framework....19
Figure 5.2: yEd Plug-In code....23
Figure 5.3: Example GML format....24
Figure 5.4: Defining Node Graphics in GML format....24
Figure 5.5: Defining Edge Graphics in GML format....25
Figure 5.6: Defining Edge LabelGraphics in GML format....25
Figure 5.7: UML Class Node in GML format....26
Figure 5.8: UML Class Node displayed by yEd....26
Figure 5.9: Association and Implementation edges in GML format....27
Figure A.1: VizzAnalyzer screenshot....31
Figure A.2: VizzAnalyzer applying the plug-in....32
Figure A.3: The plug-in configuration dialog....32
Figure A.4: The resulting diagram without layout....33
Figure A.5: yEd solving node's size....33
Figure A.6: Fit Node to Label dialog....33
Figure A.7: yEd applying layout....34
Figure A.8: The Direct Orthogonal layout configuration dialog....34
Figure A.9: The resulting Class Diagram....35

1 Introduction

Software development is a complex task. The development process typically consists out of a number of phases (among others, analysis, design, implementation and testing). Nowadays software development project are so complex that they often can not be solved by a single engineer. Therefore individuals are cooperating in teams to accomplish these complex tasks. As the activities which need to be performed are non trivial and require the sharing of complex information, the performance of the software engineering activities in every of these phases requires a high amount of communication between the involved individuals. Different formal and informal methods have been evolved to help the team members to communicate efficiently among each other and understand the software, its requirements, design and implementation.

UML [RJB04] is one language used for communicating the relevant information within a team, documenting the software. It is a collection of different kinds of diagrams, designed for modeling a software system. Class diagrams is the most common and used of this UML diagrams. They represent the static structure of the software system, depict its parts and shows how they are related.

Lifetime of software is hard to predict. Requirements change with time and the software has to be adapted to fulfill the new needs and to adapt to changed environments. According to different studies the cost for maintaining software represents between 50% and 90% of its total cost. [Fos91], [Kos03].

To decrease the maintenance cost, several researchers are been studying ways to assess and assure software quality, and thereby decreasing the maintenance costs. Among others, Lincke et. al. is researching a Standard- and Metric-Based Software Quality Model, which shall help to assess the software quality with the help of static analysis and software quality metrics [Lin07]. The designed analysis are implemented in the VizzAnalyzer tool[1] [LP05], which allows to analyze software systems for design patterns and anti-patterns, to find cycles or death code. The results of these analysis need to be communicated to the software engineers owning a particular software system in an appropriate way.

1.1 Context of the Thesis

VizzAnalyzer is a framework for reverse engineering, allowing the integration and interaction of different analysis and visualization tools. It is composed in three modules:

Retrieval module – it is responsible for extract the information from a software's source code and store it as a graph using the Common Meta-Model format.

Analysis module – it applies different metrics to the system and store the result information as properties in the same graph.

Visualization module – it provides tools to visualize the data held in VizzAnalyzer.

The currently connected visualization plug-ins allow not for a appropriate visualization of analysis results to discuss them with software engineers in a familiar and appropriate way, showing the relationship between the analysis results and the design or implementation of the software system under consideration. UML is currently the standard for the representation of software systems under development and maintenance. Therefore it would be an ideal means for representing the analysis results for communication to and discussion with other software engineers. The goal of this thesis therefore to develop a visualization plug-in for the VizzAnalyzer which allows to represent the analysis results as UML Class Diagrams.

1 http://www.arisa.se

1.2 Problem

The problem addressed by this thesis is the development of a UML visualization component for use with the VizzAnalyzer tool. This component shall be capable of visualizing any software system processed in the VizzAnalyzer. This problem is not trivial to solve, since UML itself is a complex representation having many elements and relations. However this thesis is limit to represent the data as Class Diagrams, the UML semantic is not unambiguously described and it is free to the user to define what meaning the different elements and relations have.

Additionally, the software systems are represented as graphs in the VizzAnalyzer which contain the information about a certain software system according to the Common Meta-Model introduced by Lincke in [Lin07]. A transformation from this internal representation to UML is not yet defined. More over, the generation and representation of a UML class diagram can be automated, but applying a usable layout to the diagram showing the elements in a intuitive position with few overlapping and edge crossings is another challenge.

1.3 Goals and Criteria

This section describes the goals pursued by this thesis in order to solve the problem and the criteria used for validating the goals:

- The first goal is extend the VizzAnalyzer to visualize the contained data as an UML Class Diagram. The precondition is that the contained data contains the relevant information to generate the class diagram. The goal is met when the data is displayed as classes and relationship between them in a UML 2.0 conform way. Classes shall contain the fields and methods. The fields contains type and visibility information. The methods contains parameters, return value and the type of theirs. The relationships shall correspond to the relations identified in the code and aggregate multiple relationships.

- The second goal is the generation of a layout for the output diagram, the entities and relations shall be automatically placed in the diagram. The goal is met when the classes are placed without overlapping, and the relations don't cross any class. Minimization of crossing edges and bend minimization will be desirable.

- The third goal is to allow the user to interact with the plug-in. The user should be able to decide which data is need to be shown in the diagram, this is the visibility needed to be displayed, and which relations should be generated.

1.4 Motivation

It is worthwhile to create a UML visualization component for the VizzAnalyzer since none of the currently existing visualization components can create UML conform visualizations. Yet, visualizing the analysis results in a UML conform way promises to improve the communication of analysis results and findings within the team performing the analysis and to other stakeholders. In particular software engineers, which are supposed to verify identified issues or to improve a design, will find it helpful if the information is provided in a familiar way which is similar to the used way of describing software systems, which is nowadays almost exclusively UML.

1.5 Outline

Chapter 2 presents the background knowledge necessary for understand certain parts of this thesis. It discusses the VizzAnalyzer, Common Meta-Model, Grail, yEd, and UML class diagrams. Chapter 3 describes the requirements of the visualization plug-in. This is a formalization and refinement of the goals stated in Section 1.2. Chapter 4 covers the analysis of the different approaches studied for resolve the problem. It presents different possibilities explored and finally concludes with the discussion of the solution selected. Chapter 5 describes the design and implementation of the solution selected. Chapter 6 concludes the thesis and discusses future work. Appendix A is the user manual of the plug-in developed.

2 Background

This section describes some background knowledge necessary to understand certain parts of this thesis.

2.1 VizzAnalyzer

VizzAnalyzer is a reverse engineering framework, developed at the Växjö Universitet, for the analysis and visualization of programs. It was originally developed for help researchers in the investigation and innovation in the fields of software analysis and metrics.

The process of Software analysis and visualization is shown in Figure 2.1. It has three main steps which are:

First the information is extracted from the sources files with the help of parsers, this information is retrieved as the syntax tree and contains the same information as in the sources files.

In the second step is performed a process for convert the information from the Base Model into a suitable format to perform later analysis, the Common Meta-Model described below.

Finally this information, once analyzed, needs to be displayed.

In this process the information is converted again to a suitable Model for the visualization domain.

This model needs to be finally converted to the specific format of the tool used for visualize the information.

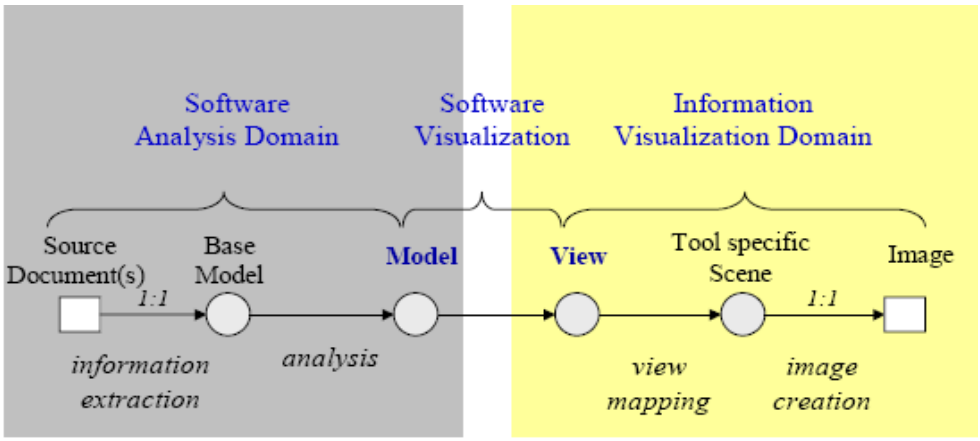

Figure 2.1: Software Analysis and Visualization Process

2.2 Common Meta-Model

The Common Meta Model (CMM) is intended to represent the software properties necessaries for calculate metrics. The model is based in the Meta-Model theory. This theory helps the definition of an abstract representation of a software system using diferent levels of abstraction: The meta-meta model is top layer, and represents the most abstract information of a software, below this layer is the Meta-Model a middle representation between the source code, and the Meta-meta model. The low down layer, is the Model, in figure 2.1 extracted from [Lin07], is shown as the Base Model, and its directly related with the source code.

2.3 GRAIL

GRAph ImpLementation is a graph library, developed at Växjö University, for manage graphs. It provides different representations as well as different algorithms and filters that can be applied to the graphs. For the context of this thesis we only use it for reading and filtering the information described in the Common Meta Model.

2.4 UML Class diagram

Traditionally a program was a list of instructions that manipulates data. With the object oriented paradigm, the program is seen as a collection of "objects" that cooperate together. Each object is capable of receiving messages, processing data, and sending messages to other objects. A class defines the abstract characteristics of the "objects", its characteristics (the data) and the things it can do (the behavior).

We use diagrams to represent the software. The class diagrams give us a graphical view of the types of objects in the system and the various kind of static relationships that exist among them. In the diagram the classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations. The relations are represented as lines with special arrows or symbols in the extremes that distinguish the different kind of relations.

The class properties (the data) has two different notations in the class diagram, it can be showed as attributes or as associations. The attribute notation describes a property as a line of text within the class box itself. The full form of an attribute is "visibility name: type multiplicity = default_value {property-string}". The association notation is a solid line between two classes, directed from the source class to the target class. The name property goes at the target end of the association, together with its multiplicity.

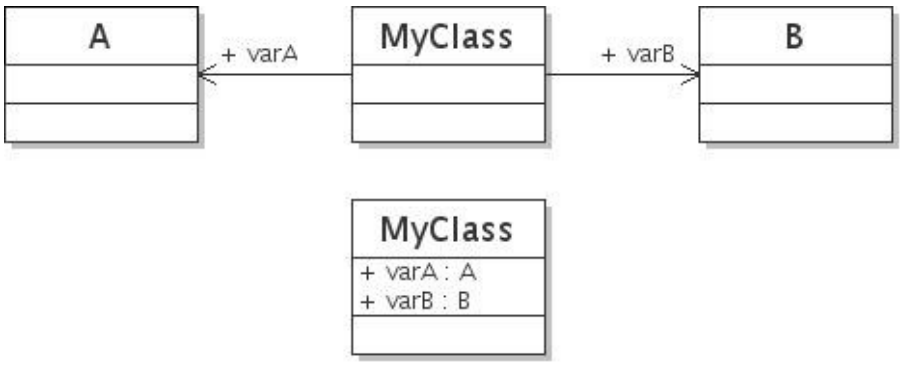

Figure 2.2: Different representations of a class property

The notation for the operations describes it as a line of text within the class box, in the operations containment. The full form of an operation is "visibility name: (parameters-list) return type {property-string}". The parameters in the parameter list are notated in a similar way to attributes, "direction name: type = default_value" and separate by ",".

The relationship between classes, can be, a part from associations, generalization, realizations and nested classes. Generalization represents the hierarchy between classes, it's represented as an arrow with a white triangle arrow head. The realization represents a class that implements an interface, and is showed as an generalization but with a dashed line. Nested classes are a relation between classes, when one class is defined inside another class, the notation for it is an arrow with a cross circle at the endpoint of the outer class.

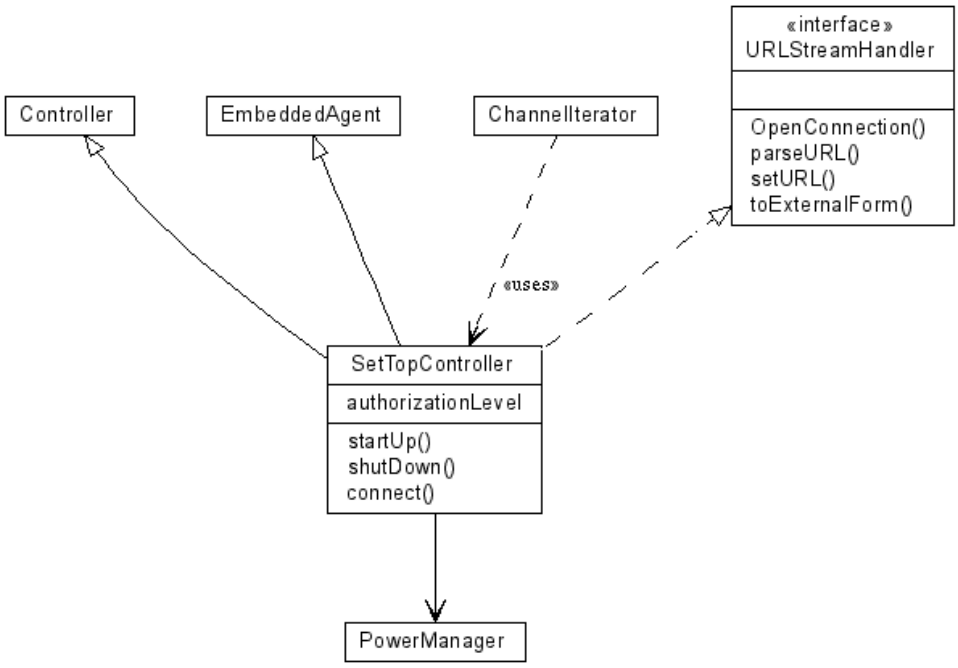

Figure 2.3: Different UML relationships

In the previous image we show the use of the relations. In that example SetTopController inherits from Controller and EmbeddedAgent (multi-inheritance) and implements the interface URLStreamHandler, also is show an association of type PowerManager.

Exist more elements in a class diagram, but they made references to concepts that involves the behavior of the system, like compositions or aggregations, or can be difficult translate into the source code, like n-ary associations. These are irrelevant for the goals of this thesis, a global overview if this elements are in [Fow04].

2.5 yEd

yEd[2] is a graph editor which is programmed in Java and uses lightweight widgets. It can be used to quickly and effectively show drawings and to apply automatic layouts to a range of different diagrams. It has a freeware license. VizzAnalyzer use it for visualize its own data. Last versions of the program allow the visualization of UML Class nodes and UML relationship edges.

2 http://www.yworks.com/products/yEd/

3 Requirements

This section describes the technical aspects of the problem and outlines the main idea for the solution. Further the users of the system, the Use Cases and the Functional Requirements, are described to analyze and document the requirements.

3.1 Users

There is only one user of the system. We call him simply User. He is the person analyzing a software using the VizzAnalyzer tool. He retrieves data form the program to analysis, applies metrics, analysis, and creates visualizations of the results.

3.2 Use-Cases

In this section we present an overview of VizzAnalyzer from the user perspective. We use Use-Cases to better understand them, for a complete view of all the Use-Cases refer to The VizzAnalyzer Handbook [PLL05].

Use-Case Model

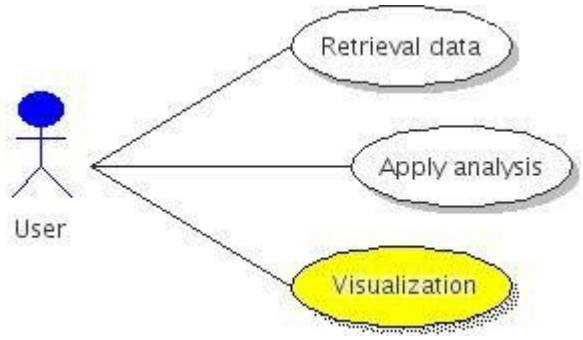

Figure 3.1: VizzAnalyzer Use-Case

Use-Case description

Use Case	**Retrieval data**
Brief description	User selects the folder where the source code of the program he wants to evaluate is found. The system retrieves the information needed from it.

Use Case	**Apply Analysis**
Brief description	User selects the current analysis he wants to perform on the data. The system applies this analysis to it.

Use Case	**Visualization**
Brief description	User selects a tool to visualize the data. The system exports the data into a format that the visualization tool can manage for display it.

UML Plug-in

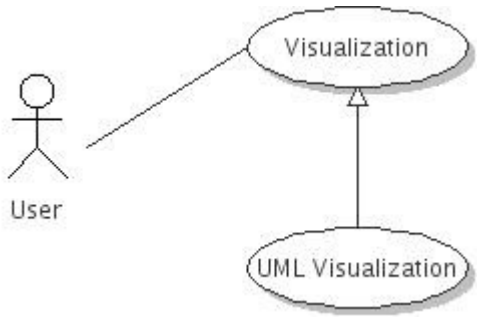

Figure 3.2: Visualization Use-Case

We extend the Visualization Use-Case from VizzAnalyzer to adapt it to the needs of the UML Visualization, we presented a detailed view of the Use-Case, showing the steps the user needs for success the scenario.

Use Case	**UML Visualization**
Brief description	User selects the current graph for visualize as UML Class Diagram.
Actors	User
Preconditions	A software system has been retrieved and is available as graph in the VizzAnalyzer. It is ready to be exported to a analysis or visualization plug-in.
Postconditions	The software system is displayed as a UML Class Diagram. 2.1ext, 5.1 ext. Display a information message to the user.
Main Success Scenario (MSS)	1. User presses Viewers → UML 2. System ask the user for the visualization options (what to visualize and how). 3. User selects the options he desires. 4. System process the data and calculates the diagram considering the selected options. 5. System presents the created diagram to the user
Alternatives	2.ext No data is selected for display. 2.1ext Systems display a information message. Exit Use-Case. 3.ext User close the options dialog. 3.1.ext Default options are applied. Return MSS 5. 5.ext Data have no the necessary information to produce the Class Diagram. 5.1ext Systems display a information message. Exit Use-Case.
Extension point	None.

3.3 Functional Requirements

In this section we list the function requirements for our UML plug-in. Each of the functional requirements has an id, brief description, pre-condition and post-condition.

Id	FR1
Description	The system should display the data as a class diagram
Pre-condition	The data is in Common Meta-Model format.
Post-condition	A Class Diagram is display with the information found in the CMM.

Id	FR2
Description	The system should not modify the data
Pre-condition	The data is in Common Meta-Model format.
Post-condition	The data is not modified.

Id	FR3
Description	The user should choose the information he wants to see in the diagram
Pre-condition	The data is in Common Meta-Model format.
Post-condition	The data is filtered with the information the user wants to display.

Id	FR4
Description	The system should display the information without overlapping of nodes
Pre-condition	A Class Diagram is calculated.
Post-condition	The Class Diagram has been layouted without overlapping of nodes.

Id	FR5
Description	The layout shall be optimized in number of cross edges and edge's bends
Pre-condition	A Orthogonal Layout is calculated for the Class Diagram.
Post-condition	The Orthogonal Layout has been optimized in minimum size of crossing edges and edge's bends.

3.4 Non-Functional Requirements

Id	NFR1
Name	Memory Consuming
Description	The contained data in VizzAnalyzer can be extremely huge, therefore is a requisite for the plug-in to minimize the size of memory needed for its correct work.

Id	NFR2
Name	Performance
Description	The visualization process shall be completed in a feasible time, meaning seconds or minutes, rather than hours or days.

4 Analysis

In this section we discuss the analysis of the different approaches studied for solving our problem and the reasons for the selected solution.

The problem proposed in the section 1.2 can be divide into two sub-problems. First, we have to convert the data from the Common Meta-Model into a Class Diagram Model, we dedicate section 4.1 for the analysis, and section 5.1 for the implementation. Secondly, we have to extract the data in the Class Diagram Model and display it in a formal Class Diagram. We dedicate section 4.2 and 5.2 for this part.

4.1 Mapping from the CMM to UML Class Diagram Model

We can see a class diagram as a graph, where the nodes are the classes or interfaces and the edges the relationships between this classes. In this step we want to define a mapping from the elements and relations of the information hold by VizzAnalyzer in the Common Meta-Model into the information needed for the class diagram, which is a UML class diagram specific meta-model.

The following figures 4.1 and 4.2 shows the definition of the Common Meta-Model that are relevant for our mapping, the complete description on the Common Meta-Model can be found in [Lin07].

```
Model       ::= Class*
Class       ::= Field* Method* Constructor*
Class           AbstractClass | AnonymousClass
Method      ::= FormalParameter*
Constructor ::=  FormalParameter*
```

Figure 4.1: Common Meta-Model productions P and specialization hierarchy

```
IsFieldOf           :  Class X Field
IsMethodOf          :  Class X Method
IsConstructorOf     :  Class X Constructor
IsActualParameterOf :  Method X FormalParameter,
                       Constructor X FormalParameter
Invokes             :  Constructor X Constructor,
                       Constructor X Method,
                       Method X Constructor,
                       Method X Method
Accessess           :  Constructor X Field,
                       Method X Field
Extends             :  Class X Class
Implements          :  Class X Class
contains            :  Class X Class
```

Figure 4.2: Common Meta-Model, binary semantic relations in R

The elements of the UML class diagram specific meta-model are classes, attributes and methods. This information will later be represented as rectangular squares that contain information about the names, and types of these elements. The relations between this classes are Inheritance, Realization, Association and Nested classes. More formal, we defined our Class Diagram model in the figures 4.3 and 4.4.

```
Diagramᵘ  ::= Classᵘ* Interfaceᵘ*
```

Figure 4.3: UML Class Model productions P and specialization hierarchy

```
Associationᵘ :  Classᵘ x Classᵘ,
                Classᵘ x Interfaceᵘ
Dependencyᵘ  :  Classᵘ x Classᵘ,
                Classᵘ x Interfaceᵘ
Extendsᵘ     :  Classᵘ x Classᵘ,
                Interfaceᵘ x Interfaceᵘ
Implementsᵘ  :  Classᵘ x Interfaceᵘ
Nestedᵘ      :  Classᵘ x Classᵘ
```

Figure 4.4: UML Class Model, binary semantic relations in R

UML allows two different notations for the properties in a class, they can be expressed as attributes inside the class, or as associations, edges between classes. For this thesis we are going to represent the properties as association if the reference class is also shown in the diagram.

The dependencies show relations of usage between classes, some of this dependencies doesn't need to be shown, for example if a class C owns a property of the type A, then the dependency between the class C and class A is unnecessary.

We define the mapping functions α as show in figure 4.5.

```
α(Class, Field, Method,        →  Classᵘ: Class.isInterface = no &
  FormalParamiter)                Field.type /∃ class*
α(Class, Method,               →  Interfaceᵘ:
  FormalParamiter)                Class.isInterface = yes
α(Class, Field)                →  Associationᵘ(Class x Field.type):
                                  Field.type ∃ (class* ∩ Class)

α(Invokes)                     →  Dependencyᵘ, iif not exists association
α(Accessess)                   →  Dependencyᵘ, iif not exists association
α(Extends)                     →  Extendsᵘ
α(Implements)                  →  Implementsᵘ
α(contains)                    →  Nestedᵘ
```

Figure 4.5: Mapping to the UML model

4.2 Visualization (of the Class Diagram)

Once we have the data as a class diagram model we can display it as a next step. For this part numerous options have been studied. They range from reusing already existing tools for drawing UML diagrams to the generation of a new tool using different options.

Our criteria for selecting the different alternatives where in particular:

1. The tool has to be free.
2. The tool should allow the automatically generation and layout of the class diagram.
3. The tool should be able to import a common format or have well defined his own file format, so we can export the data easily.

4. A new version of the VizzAnalyzer is to be integrated into the Eclipse framework. If this tool can be integrated, will be desirable for the plug-in to be also integrated into this framework.
5. The UML class diagrams shall be UML 2.0 conform.
6. The solution shall allow for a fast implementation and good results (content/detail and layout of created UML class diagrams).

We evaluated different possibilities according to our criteria. These are individually discussed in the following sections.

4.2.1 Class diagram viewer

In this section we present the different tools we were studying to use for displaying the Class Diagrams. Also we present two frameworks for the generation of diagrams. Our goal is to find a non-commercial tool, that allows displaying and layout of the Class Diagram and which meets our evaluation criteria (1 – 6).

Alternatives

Program	Rational Rose, stand-alone tool
Diagrams	Supports all kind of UML diagrams, but is not UML 2.0 conform.
Layout	Yes, hierarchical layout.
File format	Proprietary, but can import data from XMI.
License	Commercial
Description	Tool created by the original developers of the UML. However is not free, they sell academic license to universities.

Program	Omondo UML, integrated onto eclipse.
Diagrams	Is not UML 2.0 conform.
Layout	Yes, hierarchical layout.
File format	Proprietary, but can import data from XMI.
License	Freeware
Description	This tool are highly integrated with eclipse and Java. Is not possible to generate a diagram, without generating the source code of it.

Program	UMLet, stand-alone tool
Diagrams	Supports all UML 1.0
Layout	No
File format	Use XML the description is available.
License	GPL
Description	This tool is designed to draw fast UML sketch. Is more a painting tool that a UML diagram tool.

Program	yEd
Diagrams	Generic tool for drawing graphs.

Layout	Yes, hierarchical, orthogonal, directed orthogonal.
File format	Standard GML file.
License	Freeware
Description	yEd is a generic graph visualization program. It allow several layout algorithms, and with some trick we can represent Class diagrams, using HTML.

Frameworks

Program	Violet, stand-alone tool, a plug-in for run onto Eclipse is available.
Diagrams	Framework for graph diagrams, it has already developed a Class Diagram tool.
Layout	No
License	GPL
Description	Tool created by Cay Horstmann, author of several Java books, as example of framework. It's really simple to use and extend, the code is well organized and understandable.

Program	Eclipse Graphical Editing Framework (GEF)
Diagrams	GEF originally comes without any diagrams. Exist a project for display UML using this framework called UML2 Tool, but is still under development.
Layout	No
License	Eclipse Public License.
Description	Framework developed for the Eclipse team, to extend Eclipse allowing diagrams generation.

Mapping Table

Crit \ Tool	Rational	Omondo	UMLet	yEd	Violet	GEF
1	no	yes	yes	yes	yes	yes
2	yes	yes	no	yes	no	no
3	yes	no	yes	yes	yes	yes
4	no	yes	no	no	yes	yes
5	yes*	yes*	yes*	no	yes	yes
6	yes	no	no	yes	yes	no

- *The tool is not total UML 2.0 conform, but missing parts are not needed for the plug-in.*

Omondo have ruled out because we can not import the file, also yEd was out because was not UML conform. None of the other free UML tools allowed us to automatically layout our resulting Class Diagram. This made it necessary to develop our own layout algorithms, or try to find another solution.

First it seemed most promising to create our own tool reusing the Violet framework, for drawing the UML class diagram, and then to implement our own layout algorithm. After some prototyping and further study of literature related to layout algorithms it turned out, that this was a very complex task.

We finally decided follow another alternative which was previously out-ruled, since it did not directly support the creation of UML class diagrams, but it had already very efficient layout algorithms implemented. We now focused on selecting yEd as our diagram tool. Since it is not intended for UML diagrams we needed now to find a way to work around this issue. This will be discussed in detail in Chapter 5.

4.2.2 UML Layout

UML Layout problem has becoming of special interest in the last years. Different studies work on extending the traditional graph or tree layout algorithms for satisfying the needs of providing a good layout for UML Class Diagrams. Actually there are two main directions in the studies. The hierarchical approach and the directed orthogonal approach. We are going to present them, and describe briefly the basis and problems found in it.

Hierarchical approach

Hierarchical approach is based in the Sugiyama algorithm [Su86]. This algorithms are based on the common divide and conquer technique, which is used for drawing acyclic directed graphs. Different variants have been developed for drawing other kinds of graphs.

The main idea of the Sugiyam algorithm is to divide the nodes into layers, in a way that nodes of a layer L+1 have not outgoings edges to layers in level L or lower and after that, the algorithm reorders the nodes in each layer, trying to minimize the number of crossing edges with the precedent or antecedent layer. Refer to [Su86] for a more detailed description.

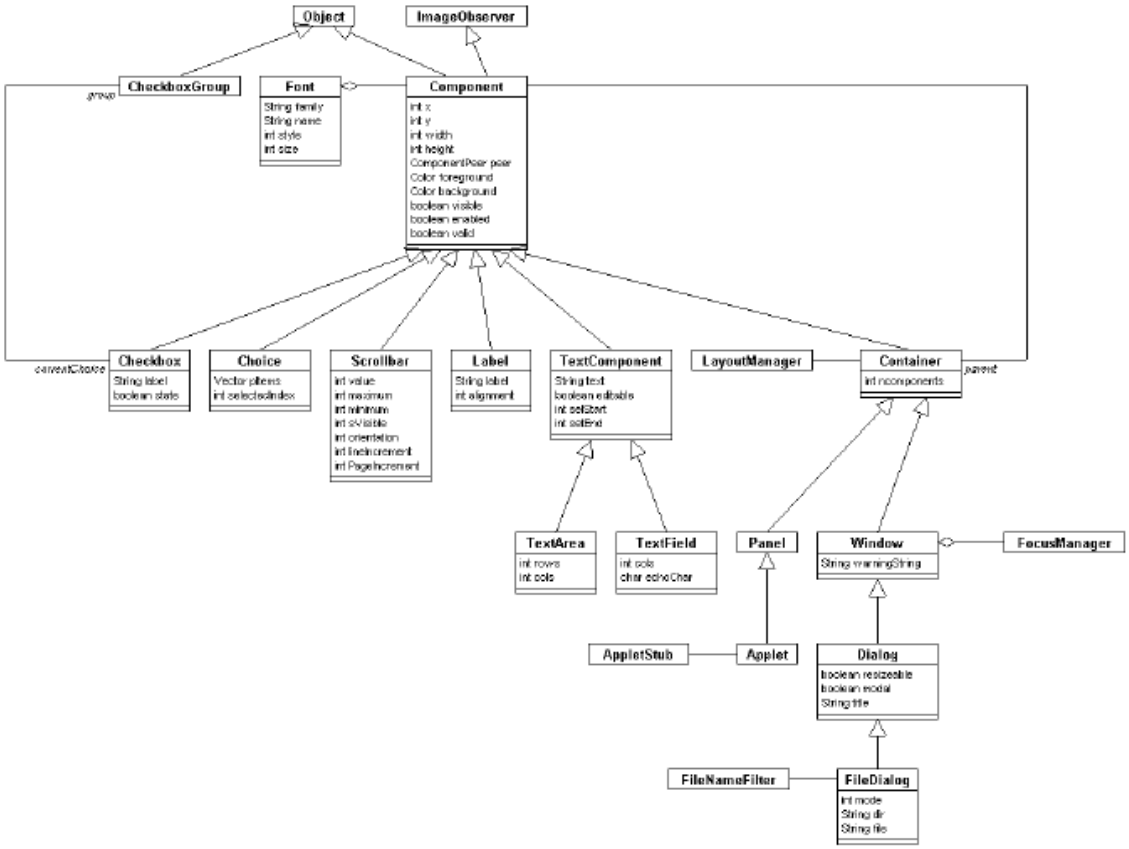

Figure 4.6: UML Hierarchical Layout

In [Sem97], Jochen Seeman proposes an for UML adapted version of the Sugiyam algorithm. This UML layout algorithm has two main steps. First, it calculates the layout of a subgraph of the class diagram, formed only for the classes, and the inheritance and implements relations, using the original Sugiyama algorithm. The second step is to incrementally add the rest of the classes and relations, trying to minimize the size and crossing edges. Refer to [Sem97] for the detailed algorithm. See Figure 4.6 for an example.

Hierarchical approach works fine when the number of inheritance is high. But for diagrams with a low number of inheritance or high number of associations the results are poor. Würzburg University researchers are currently extending this layout algorithm, in the SugiBib project, trying to fix this limitations. It seems that after they finish the research, the university will distribute the source code.

Directed Orthogonal approach

This approach is based in a variant of the Orthogonal Layout. It is a bit more complex than the the Hierarchical layout. It is fundamental to find a planar representation of the graph. First the graph has to be planar, if the graph is not planar, then the more complete subgraph is calculated. The rest of edges are temporally save. After that a planar representation is calculated for the graph, and added the edges removed trying to reduce the cross size. A orthogonal process is realized, this is place the edges horizontal or vertical in the draw, and finally the result is optimized in size and size of bends.

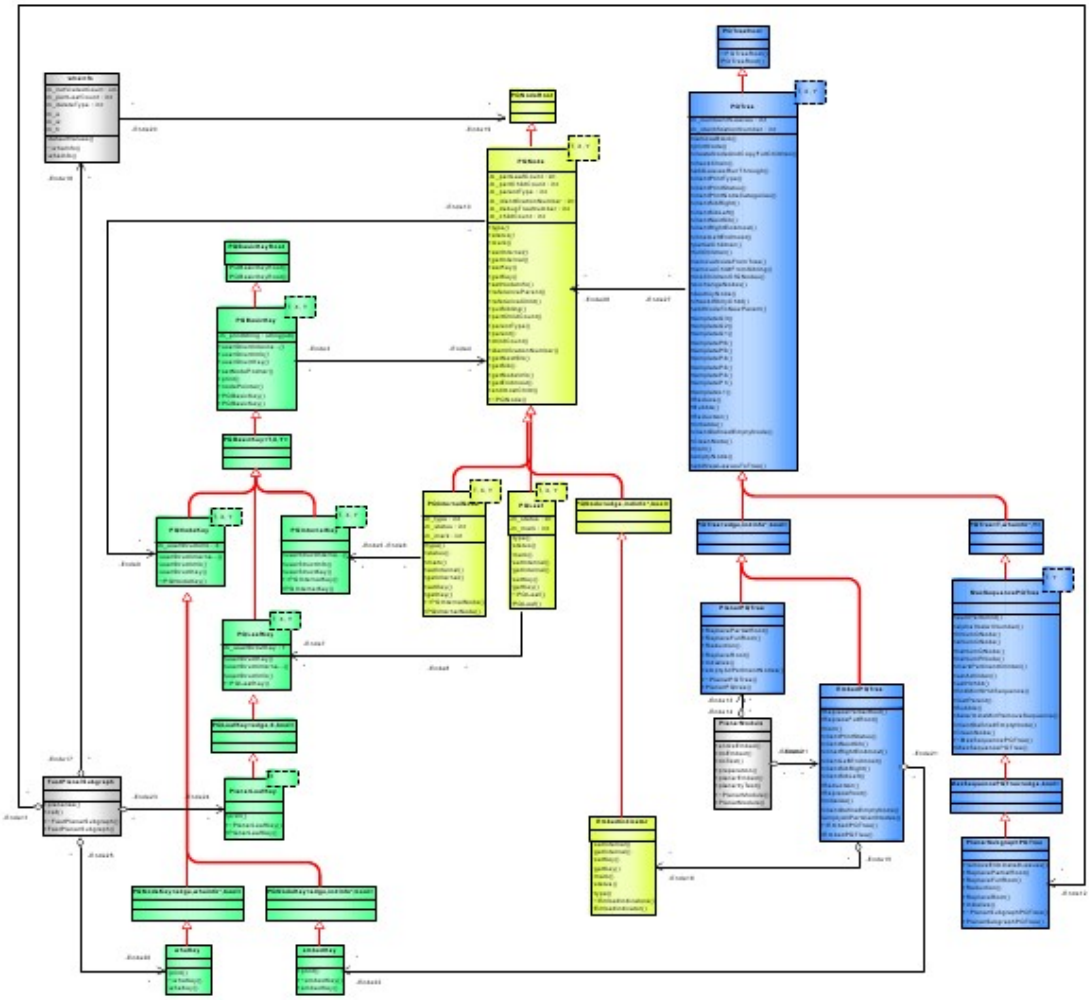

Figure 4.7: UML Directed Orthogonal Layout

The Directed Orthogonal approach presented in "A new Approach for Visualizing UML Class Diagrams" [GJK03], uses the Orthogonal layout but applies new constraints when calculating the planar representation: uniform direction within each class hierarchy and no nesting of one class hierarchy within another. And at the end, the layout merges the implementation and extension edges at the target. See Figure 4.7 for an example.

Various test implementations realizing this algorithm show that it offers better layouts than the hierarchical approach in almost all the cases. Except when the classes are highly connected by inheritance.

4.3 Discussion

VizzAnalyzer and Common Meta-Model are tools generated at Växjö University for research, they are still in development. Few third party plug-ins have been developed. Therefore, currently, no component exists that transforms the Common Meta-Model used by VizzAnalyzer to a Class Diagram Model. That's the reason we finally decided to implement our own CMM to Class Diagram transformation tool.

For the visualization part is different, Class Diagrams are very extended nowadays, and there are lot of tools intended for generation an visualization of UML. Most of this tools are intended for a manual generation of the diagrams, and few of them allow automatically layout, unfortunately none of them are free.

We tried to use one of this open source UML program and implement by ourself the layout algorithm. We intended to implement the hierarchical approach, however is not the best solution, the algorithm is easier than the orthogonal approach. We realized that implementing the layout algorithm is not a simple task requiring to implement additional algorithms for manipulate the graph necessary for the layout algorithms. This required time was outside the scope of this thesis.

The objective of this thesis is visualize the information held in VizzAnalyzer as a UML class diagram. We do not need a complex tool for manipulate the diagrams, we only need to view the diagrams. Since a class diagram can be seen as a graph, a tool for visualizing and layouting graphs can be used if it allow special draw of nodes and edges.

Currently VizzAnalyzer use yEd as one of his main visualization tools. This tool allows the insertion of some UML drawings and also offers the Directed Orthogonal layout. With a little trick, we can simulate a class node by using the horizontal rule <*HR*> tag in HTML, and thus make the normal nodes appear like UML class nodes, see section 5.3.4. Because of this option and the powerful layout algorithm we finally selected to follow this approach, since it was in the end the most promising.

5 Design & Implementation

This section covers the design and implementation issues of the plug-in. First we presented the main aspects from VizzAnalyzer architecture that are related with the Visualization Plugin. Furthermore we presented the most significant parts of the plug-in, the transformation between the Common Meta-Model to the Class Diagram Model, and the export process. In this last part we presented the GML format, used by the yEd tool to load and display the graph.

5.1 The VizzAnalyzer Framework

This section provides background on the VizzAnalyzer architecture. We explain the parts of the architecture that are reused for the Visualization plug-in. Finally, we explain how the internal data of the software system under consideration is stored and manipulated.

5.1.1 Architecture

The VizzAnalyzer Framework is a composition system. It consist of the framework-core, converters, and different wrappers connecting the plug-ins. The framework-core is a controller, it's the responsible for communicating information between the different components. The converters are reusable components, connected to the core. They provide the main functionalities. GRAIL is one of this components. The wrappers allow the connection of arbitrary reverse engineering components with the framework. e.g., Recoder or yEd. The Figure 5.1, extracted from [Lin07], shows a picture of the framework.

The wrappers are the ones we are interested in, they allow to extend the framework. They allow three variation points: retrieval, analysis and visualization.

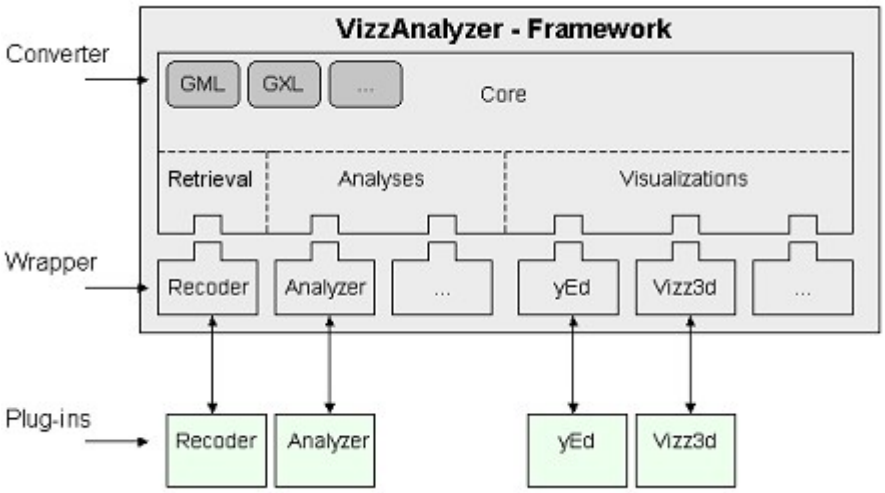

Figure 5.1: VizzAnalyzer Framework

The variation points are technically realized as a directories containing the plug-in classes. These plug-ins extend from predefined classes provided by the framework, when the framework starts, it reads this directories and tries to load the plug-ins. If the plug-in is recognized by the framework, then a new menu entry is created in the framework and is mapped to run the plug-in. Refer to [PLL05] for more detail view on the VizzAnalyzer Framework.

For connecting our visualization plug-in, the yEd graph editor, we need to extend the existing yEd plug-in to utilize the proper wrapper converting the data provided as Common Meta-Model to the UML Class Diagram specific meta-model.

5.1.2 GRAIL and the Common Meta-Model

Grail is a graph library used by VizzAnalyzer to represent the internal data. This representation consist of an annotated graph instantiated from GRAIL, where each entity (nodes, edges and the graph itself) has a set of properties associated to it. These properties are represented as objects.

The property Type of each node (or edge) represents an entity (or relation) of the Common Meta-Model. Grail allow us easily to retrieve the information stored in the properties.

Grail also offers functions for navigating the graph entirely or a subset of it using views. In this views we can specify the type of the objects (node or edges) we are interested, we can filter it depending the values of its properties. In these views we can also specify edge propagations. This means, if children of the current node are filtered but have relationship to other nodes, this relationship can be propagated to the parent node, until they are attached to a non-filtered node.

5.1.3 The Visualization plug-in

The plug-ins are Java compiled classes that extend from the *PlugIn* abstract class given by the core and implement the *PlugInInterface*. This *PlugIn* class defines an *ActionList* property. This Action is used for the VizzAnalyzer to call the execution of the plug-in. Therefore the plug-in, when initializing, needs to add the code the action is going to execute.

When the VizzAnalyzer framework is starting, it searches the predefined directories, retrieval, analysis and visualization looking for the wrappers. Once they are found, those wrappers are constructed and instantiated with a call to the *initPlugIn()* method.

The application is running waiting for the user to interact. When an interaction occurs the GUI calls the correct plug-in for execute the Action.

The information exchange between the wrappers and the framework is handled via the *setGraph()* and *getGraph()* methods from the *ProjectManager*. All program information, as discussed before, is internally handled as GRAIL's graphs.

5.2 Transformation process

As we described in the previous chapter, in the transformation process we are going to map and export the data from the Common Meta-Model to our Class Diagram Meta-Model. Therefore we need to traverse the graph representing the data and retrieve the information needed in the diagram.

We realize this transformation in two steps. First, we iterate over a view of the graph which contains only the nodes relevant for the UML Class Diagram Meta-Model. The other nodes are filtered. This will let's us know the classes participating in the diagram and the class properties and relations, e.g., associations or attributes.

Secondly, for each class we are going to display in the diagram, we look for all the available information needed in a class diagram. We use different views of the Common Meta-Mode to archive this task.

Name	**Class View**
Elements	*Class*
Relations	*Extends* *Implements* *Contains*
Description	The mapping between the Common Meta Model and our Class Diagram Model is directed, we use for: • Nested classes: *Contains* edges. • Generalizations: *Implements* edges. • Realizations: *Extends* edges.

Name	**Field View**
Elements	*Class* *Field*
Relations	*IsFieldOf*
Description	The field view gives information about the properties of a class. As mentioned before, we look the type property of the *Field* for know when this property have to be shown as an attribute or as an association.

Name	**Use View**
Elements	*Class* *Constructor* *Method*
Relations	*Invokes* *Accessess*
Description	In the use view we found all the calls and properties accesses from the current class. We have to filter this result with the results obtain in the field view to retrieve the information we finally want to show in the class diagram.

Name	**Method View**
Elements	*Class* *Method* *Constructor* *FormalParameter*
Relations	*IsMethodOf* *IsConstructorOf* *IsFormalParameterOf*
Description	This view allows us to fill the methods compartment in a class. Given a class element, its outgoing edges are the methods (*Method* and *Constructor* elements in the CMM) we wanted. And for each method looking again its outgoing edges we reach the parameters (the *FormalParameter* in the CMM). We look at the position property in the *FormalParameter*, for knowing in which order we have to display it. After retrieving this information we create the string following the UML convention, we look the *Method* properties for retrieve the visibility and the static and abstract attributes, and write the pertinent HTML tags, to make the text underlined or in italics.

Before starting the loop for the classes, we instantiated this views. Later for each class element in the Common Meta-Model we ask the different views for its relations. We need to filter this results for avoiding redundant information, e.g., ignoring usage relation between the classes when there already exist an association between them.

5.3 Visualization process

Finally we are using yEd for display the UML Class Diagram. The VizzAnalyzer already uses yEd as one of its visualization plug-ins. Therefore we are going reuse the existing plug-in and adapt it to our needs.

5.3.1 VizzAnalyzer yEd plug-in

As mentioned in Section 5.1.3, the plug-in is a Java class which extends from *PlugIn* and implements the *PlugInInterface*. The *PlugIn* class contains the code needed for the framework for adding the plug-in to a menu entry in the user interface. The *PlugInInterface* defines the *initPlugin* and *toString* methods we need to implement. The initPlugIn will be called by the framework when loading the plug-in. The toString is used to display the plug-in name onto the menu in the GUI.

```
01 public class yEd extends PlugIn implements PlugInInterface {
02
03   private static final String YED_PROGRAM_DIR
         = Data.myViewPlugInDir + "/yEd/bin/";
04   private static final String YED_PROGRAM_FILE = "yEd.jar";
05
06   public void initPlugIn(ProjectManager pmP) {
07      this.pm = pmP;
08
09      Action yEdAction = new AbstractAction("start", null) {
10         public void actionPerformed(ActionEvent ae) {
11            run();
12      }};
13      actionList.add(yEdAction);
14   }
15
16   public void run() {
17      GraphInterface g = pm.getSelectedGraph();
18      if (g == null) return;
19      GML gml = new GML();
20      gml.toGML(new File("_test.gml"), g, true);
21      Runtime runtime = Runtime.getRuntime();
22      String[] cmd = { "java", "-Xmx256m", "-jar",
            YED_PROGRAM_DIR + YED_PROGRAM_FILE, "_test.gml" };
23      runtime.exec(cmd);
24   }
25
26   public String toString() {
27        return "yEd";
28   }
29 }
```

Figure 5.2: yEd plug-in code

Within the *run()* method (Figure 5.2, lines 16-24), the plug-in exports the internal graph into a temporary GML file (lines 19-20), and after runs the yEd program with the file as argument to display it.

For our visualization plug-in we want to display the Class Diagram we are generating. Therefore, we are going to change lines 19-20 from the original yEd plug-in, to export our Class Diagram Model we calculated in the previous section.

The exporting file will be also a GML format, due some technical issues with the current GML converter, proportioned by VizzAnalyzer, it doesn't allow the selection of shapes in the resulting file, we are going to implement our own GML converter.

5.3.2 GML File Format

The GML File Format was designed to be a standard graph file format for easily exchange data graphs between applications. It is very extensible and allows for each program to write its own data.

The file is made up of pairs of a key and a value. The key is a string literal and the values can be integers, floating point numbers, strings, records and lists, where the latter two must be enclosed in square brackets.

The key idea behind GML is that there are some standard keys like graph, node and edge, and anybody is free to add its keys to add specific information.

```
graph [                    Defines a new graph
    directed 1             Is a directed graph, 0 if not

    node [                 Defines a new node
        id 1               This node has the id 1
        label "Node 1"     Label for the node
    ]
    node [                 Defines a new node
        id 2               This node has the id 2
        label "Node 2"     Label for the node
    ]
    node [                 Defines a new node
        id 3               This node has the id 3
        label "Node 3"     Label for the node
    ]
    edge [                 Defines a new edge
        source 1           Source is the node with the id 1
        target 2           Target is the node with the id 2
    ]
    edge [                 Defines a new edge
        source 2           Source is the node with the id 2
        target 3           Target is the node with the id 3
    ]
]
```

Figure 5.3: Example GML format

Figure 5.3 shows a simple directed graph expressed in the GML format. The italic sentences on the right are comments and don't belong to the file. For a more detailed description on the file format refer to [Him96].

5.3.3 yEd extension of GML format

yEd defines two records for storage the information needed for displaying the diagrams. These are graphics and LabelGraphics. Both can be applied to nodes or edges, but they have different properties. We are only showing the properties that are interest in the plug-in.

For the nodes, the graphics is defined as:

```
Graphic [
    x          float value, ex: 12.5
    y          float value, ex: 12.5
    z          float value, ex: 12.5
    w          float value, ex: 12.5
    h          float value, ex: 12.5
    type       shape, string value ex: "rectangle"
    fill       color, string value "#FF00AA"
    outline    color, string value "#FF00AA"
]
```

Figure 5.4: Defining Node Graphics in GML format

Where type can be "rectangle", "ellipse", "roundrectangle". For the edges the graphics is defined as:

```
graphics [
   fill          color, string value ex: "#000000"
   style         line style, string as "dashed"
   targetArrow   arrow style, string as "standard"
]
```

Figure 5.5: Defining Edge Graphics in GML format

Where style can be either "normal" or "dashed", and the targetArrow can be "none", "standard", "white_delta", "diamond". By default, if nothing is indicated, yEd places the edges labels in the center. If we want to modify this behavior, we need to write the LabelGraphics record with the following values.

```
LabelGraphics [
   text              Same name as the label
   model             "six_pos"
   position          "thead"
]
```

Figure 5.6: Defining Edge LabelGraphics in GML format

This information is not complete. There exists no official information about the records, we extracted the information generating some graphs with yEd and exporting to GML file. There is no guarantee this will remain as is now. Also for the types in the shape, line and arrow, the values shown are the ones that are useful for the plug-in.

5.3.4 Export the file

Memory consumtion is one of the main requisites for the plug-in. This is the reason why we are going to export the Class Diagram Model during its conversion. As we commented in the transformation section, first we traverse all the graph deciding which class nodes are going to be included in the class diagram file.

For each of these classes, we retrieve all the information needed. As discussed in the background chapter, a class in UML is represented as a box with three containments for the name, attributes and methods. As yEd has non proper representations available in its free version, we simulate the UML Class representation by drawing the node as rectangle, and writing the label using HTML. This allows writing horizontal lines, for separating the containments simulating a class node, see Figure 5.8. The follow example shows how a class in the GML file looks.

```
node [
id      1
label   "<html>
        <center><b>AbstractTest</b></center>
        <hr>
        - list : ArrayList
        <hr>
        + <i>TestAbstract() : void</i>
        <br>
        + <u>TestStatic() : void</u>
        <br>
        + TestFinal() : void
        <br>
        </html>”
]
```

Figure 5.7: UML Class Node in GML format

When there is not information about how to draw the node, yEd by default, displays it as a rectangular shape. Figure X shows the result displayed by yEd of the previous example.

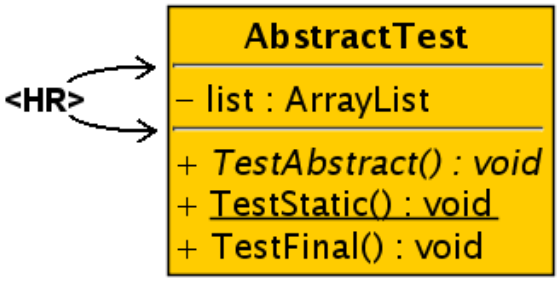

Figure 5.8: UML Class Node displayed by yEd

One time we retrieve all the information of a class, and represent it as HTML, we store it into the file instead of holding it in the memory. Only the relationships between the classes need to be temporally stored in memory, because in the GML file, the nodes need to be written before the edges.

After all the classes are written into the file, we start writing the relationship (the edges) with the appropriate style and arrow. We only modify the label position on the associations, for displaying it in the target instead of the center of the edge. The following code shows an association and a implementation relationship.

```
edge [
   source  2
   target  3
   label   "association"
   graphics [
      fill         "#000000"
      targetArrow  "standard"
   ]
   LabelGraphics [
      text         "association"
      model        "six_pos"
      position     "thead"
   ]
]
edge [
   source  1
   target  3
   label   "implementation"
   graphics [
      fill         "#000000"
      targetArrow  "white_delta"
      style        "dashed"
   ]
]
```

Figure 5.9: Association and Implementation edges in GML format

6 Conclusion and Future work

This last chapter reflects on the results of this thesis, and points out the directions of future work. First, we show how the problem presented in the introduction has been solved and we show that the goals has been achieved, criteria met. Furthermore, we show possibilities for future work.

6.1 Conclusions

The overall goal of this thesis was provide VizzAnalyzer with a visualization plug-in for displaying the a software system analyzed in the VizzAnalyzer as UML Class Diagram. To operationalize this problem we defined three goals, the plug-in has to meet to solve the problem.

- The first goal is to extend the VizzAnalyzer to visualize contained data as an UML Class Diagram. The precondition for this goal is that the contained data contains the relevant information to generate the class diagram. The goal is met when the data is displayed as classes and relationship between them in a UML 2.0 conform way. Classes shall contain the fields and methods. The fields contains type and visibility information. The methods contains parameters, return value and the type of theirs. The relationships shall correspond to the relations identified in the code and aggregate multiple relationships. We met this criterion by defining a UML Class Diagram Meta-Model and a mapping from the Common Meta-Model to it. We discussed on the Common Meta-Model and the UML Class diagrams in chapter 3, furthermore in section 4.1 we analysis the conversion between the models, and in section 5.2 we provided details on the implementation.
- The second goal is the generation of a layout for the output diagram, the entities and relations shall be automatically placed in the diagram. The goal is met when the classes are placed without overlapping, and the relations don't cross any class. Minimization of crossing edges and bend minimization will be desirable. For meeting this goal we studied several UML tools. The description and results can be found in section 4.2. After evaluating different approaches, we finally decided for a solution using a generic graph program yEd for displaying and layouting the UML class diagrams. Details of the implementation of the plug-in performing the conversation and execution of yEd, including the export to the exchange file are provided in Section 5.3. Our combined approach of using a UML conform representation and powerful layout algorithms from a 3rd party tool meets this goal.
- The third goal is to allow the user to interact with the plug-in. The user should be able to decide which data is need to be shown in the diagram, this is the visibility needed to be displayed, and which relations should be generated. This goal was met by using the plug-in configuration and the VizzAnalyzer tool. The plug-in configuration allow the user to select the information he wants to display. The user can choose the visibility level for the objects (methods and attributes) to be displayed aswell the type of relationships to represent in the diagram. The VizzAnalyzer tool also provides the necessary options for manipulate the internal data, allowing the users to filter the information the way they shall be displayed.

6.2 Future Work

Our implementation of the UML plug-in provides us the expected result and has solved the problem. Is still some work left, since the current definition of the Common Meta-Model allow not some properties needed in the Class Diagram, there is missing the isAbstract and isStatic information on the properties. We take this in consideration when writing the plug-in and when this properties will be available should not be a problem for add it to the plug-in.

Future research can be made in the associations relationships. Actually there is no multiplicity field in the Common Meta-Model. This information should be possible to extracted from the application code, but is mostly dependent from the programming language used, further research can be made to extract this information.

Another future research can be made in the layout algorithms. yEd provide us with suitable layouts that solves our problem, but this layouts are generic. Some aesthethics constraints considered in [GJK03] are not implemented, e.g. avoid nesting of one class hierarchy within another, display related class by inheritance together.

7 References

[Fos91] J.R. Foster. *Program lifetime: A vital statistic for maintenance*. In Proc. 7th IEEE Conf. Software Maintenance, October 1991.

[GJK03] Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., and Mutzel, P. 2003. *A new approach for visualizing UML class diagrams*. In Proceedings of the 2003 ACM Symposium on Software Visualization (San Diego, California, June 11 - 13, 2003). SoftVis '03. ACM Press, New York, NY, 179-188.

[Him96] Michael Himsolt. *GML: Graph Modelling Language*. December 1996. http://www.infosun.fim.uni-passau.de/Graphlet/GML/

[Kos03] Koskinen. *Software Maintenance Costs*. 2003. (based on a survey of other MIS-oriented surveys) http://www.cs.jyu.fi/~koskinen/smcosts.htm

[Lin07] Rüdiger Lincke. *Validation of a Standard and Metric-Based Software Quality Model*. Reports from MSI no. 07046. 2007. Växjo University.

[LP05] Welf Löwe and Thomas Panas. *Rapid Construction of Software Comprehension Tools*. International Journal of Software Engineering and Knowladge Engineering, 15(6):905-1023, December 2005

[Fow04] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley , third edition. 2003

[PLL05] Thomas Panas, Rüdiger Lincke, Welf Löwe. *The VizzAnalyzer Handbook*. Technical Report. October 2005. Växjö University.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.

[Sug86] K. Sugiyama, S. Tagawa, and M. Toda. 1981. *Methods for visual understanding of hierarchical systems*. IEEE Trans. Syst. Man Cybern., SMC-11(2):109-125, 1981.

[Sem97] Seemann, J. 1997. *Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams*. In Proceedings of the 5th international Symposium on Graph Drawing (September 18 - 20, 1997). G. D. Battista, Ed. Lecture Notes In Computer Science, vol. 1353. Springer-Verlag, London, 415-424.

Appendix A - User Manual

This chapter contains the user manual of the plug-in. It explains the installation and usage of the plug-in for display the analyzed software system as UML 2.0 Class Diagrams.

A.1 System Requirements

The plug-in should run on every system where JVM 1.5 is available. For efficiency a minimum of 512MB of RAM is recommended. More memory might be required depending on the complexity of the system to visualize.

The software needed for the correct work of the plug-in are VizzAnalyzer version 1.0.11 or higher, available at *www.arisa.se*, and yEd version 2.4.0 or higher, available at *www.yfiles.com*.

A.2 Installation

For the installation this instructions assume that VizzAnalyzer tool and yEd has been installed and are ready for use, if not please consult the Installation section in the VizzAnalyzer user manual. The plug-in is distributed in the *UML_plugIn.zip* file, for installing it extract the file inside the *plugIns* folder in the root directory of the VizzAnalyzer tool.

A.3 UML plug-in

For the visualization of a program two preconditions need to be fulfilled. First a graph representing the analyzed program complying to the Common Meta-Model (CMM) needs to be loaded. Then the interesting parts of the program which are going to be visualized need to selected as input.

A.3.1 Loading a graph

Prior to apply the visualization, the CMM file needs to be loaded into the VizzAnalyzer tool. Actually the only CMM files available are produced by the Retrieve Ecliplse plug-in.

For loading a file into the VizzAnalyzer tool select *Load* in the *File* menu. In the Open dialog browse the folders and select a GML file. Pressing the *OK* button makes VizzAnalyzer to load the file, in the left panel of the application the a new Data graph is show.

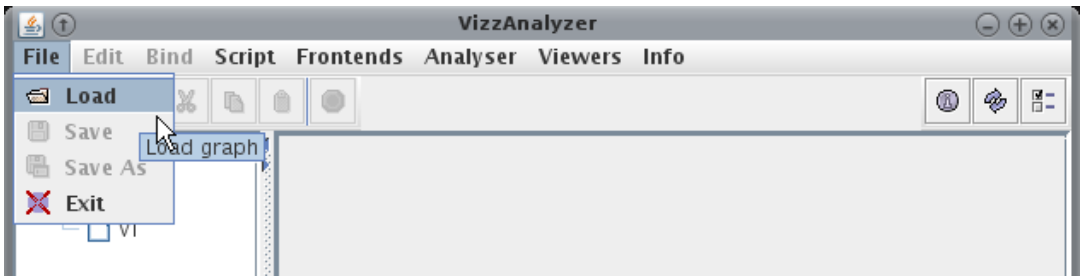

Figure A.1: VizzAnalyzer screenshot

A.3.2 Run the plug-in

Before starting the plug-in make sure a graph in the left panel is selected. The plug-in is started by selecting *yEd – Uml* in the Viewers menu, figure A.2 shows a screenshot.

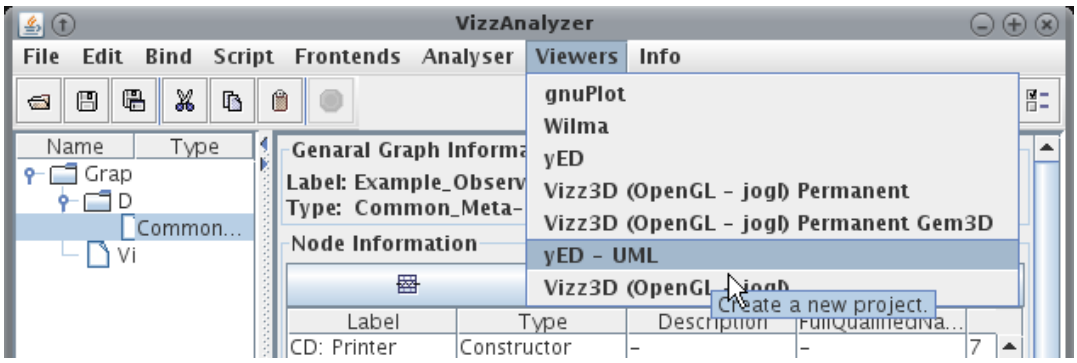

Figure A.2: VizzAnalyzer applying the plug-in

If all works correctly the configuration screen appears. Otherwise an error message appear showing the problem, the common errors are no graph was selected, return to previous step, or problems with yEd, make sure yEd is installed in the specific place.

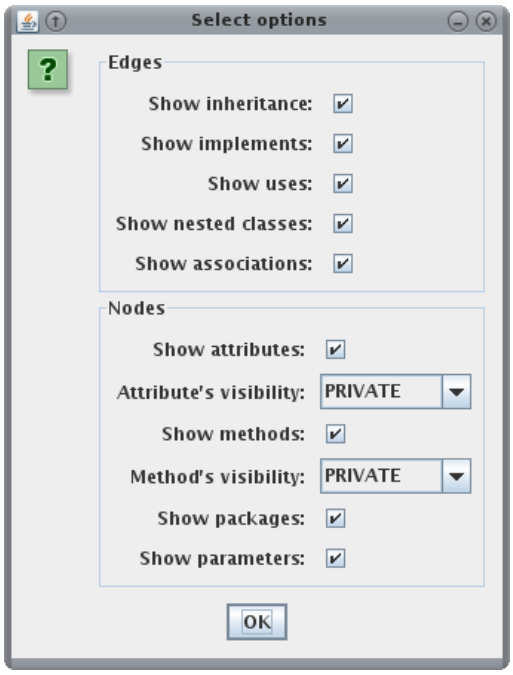

Figure A.3: The plug-in configuration dialog

The configuration screen allows the selection of different attributes for the resulting diagram. Allowing to remove some kind of relations or remove some information in the classes making its display smaller and then easy to read the resulting diagram.

The first panel are concerned in the kind of relations to be displayed, checking them makes the plug-in to calculate the specific relation determined in the label.
The other panel affect the information displayed in the classes. The visibility options allow the selection of a minimum visibility needed by a element to be displayed in the class. Finally enabling show packages, for each type, the qualified name is displayed.

After selecting the options that are interesting for the diagram pressing OK makes the plug-in calculate the ouput diagram and launch the yEd tool visualizing it.

A.4 yEd Result

This section contains a short tutorial on yEd, describing the actions needed for correct the display of the diagram. Two steps are performed, first the correction of the node size and second apply a layout to the diagram.

A.4.1 Resize nodes

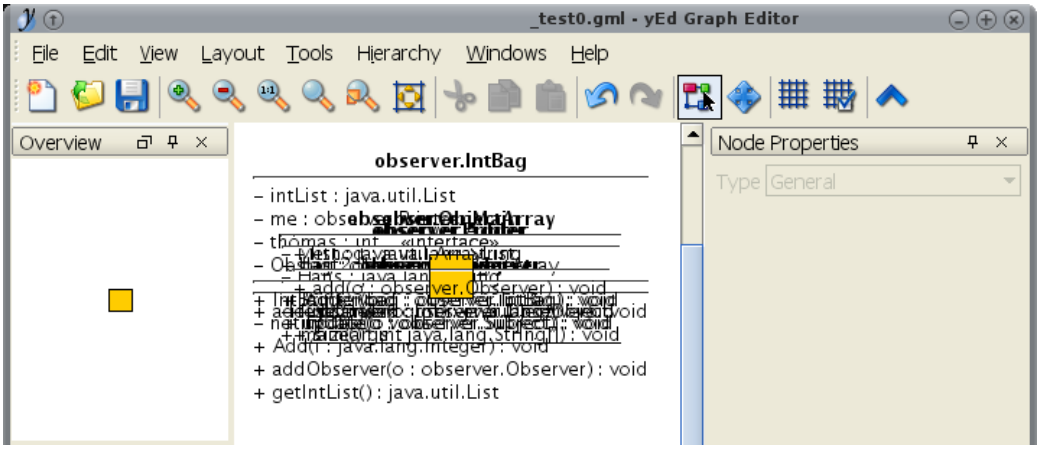

Figure A.4: The resulting diagram without layout.

As seen in the Figure A.4 the resulting diagram from the plug-in does not calculate any dimensions or position for the nodes and relations. It is necessary to use the features of yEd to realize this.

The first step is to expand the nodes to get the proper size of the labels, making them appear as real class nodes. This step is realized by selecting *Fit Node to Label* in the *Tools* menu.

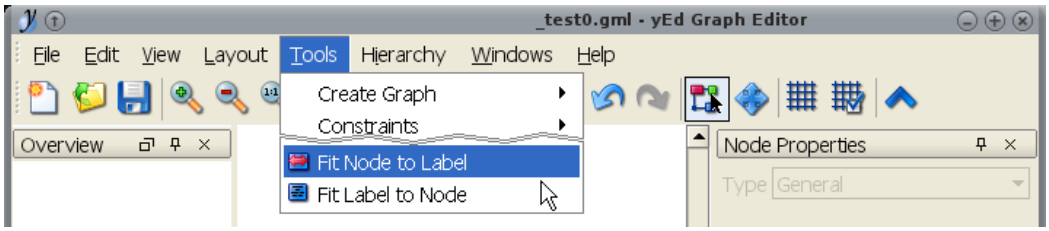

Figure A.5: yEd solving node's size

The program shows a new dialog allowing the configuration of some parameters of the tool. Selecting 1 pixel in *Vertical* and *Horizontal Space* shows a more realistic appearance.

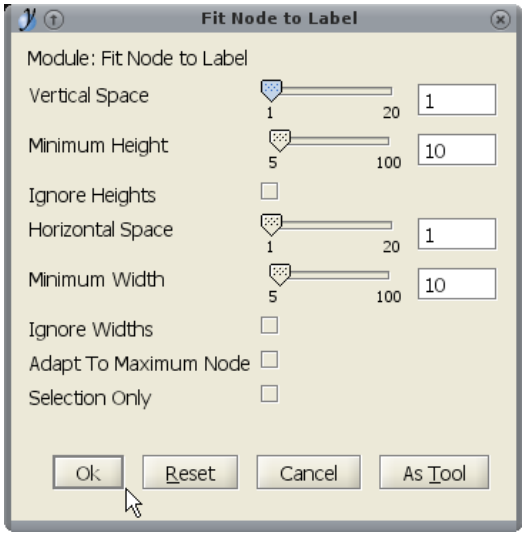

Figure A.6: Fit Node to Label dialog

A.4.2 Apply layout

The last step is to apply a layout to the diagram. yEd allows several layout algorithms. Using the Direct Orthogonal Layout gives better results. The layout is started by selecting *UML Style* in the *Orthogonal* submenu inside the *Layout* menu.

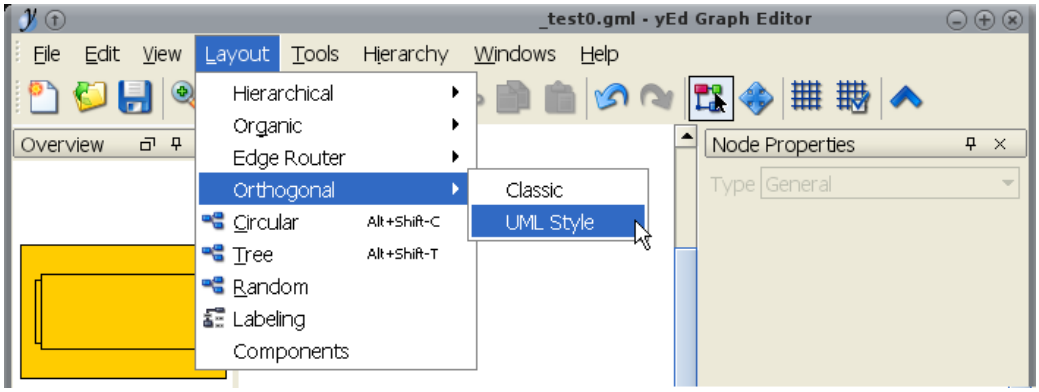

Figure A.7: yEd applying layout

Copying the same options as shown in Figure A.8, *Orientation*: *Bottom to Top*, *Criteria*: *Target Arrow* and *Target Arrow* the one with a white triangle makes the layout to place the hierarchical classes from top to down, the way is frequently used in the UML Class Diagrams.

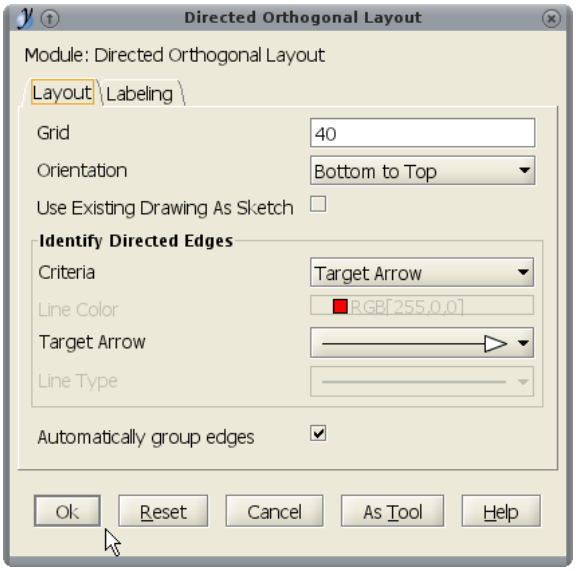

Figure A.8: The Direct Orthogonal layout configuration dialog

Pressing the OK button makes yEd to calculate the selected layout and apply it into the digram.

A.4.3 Final result

After that the resulting diagram can be manipulated, using the cursor, if necessary. The Figure A.9 shows the final result of the generated diagram.

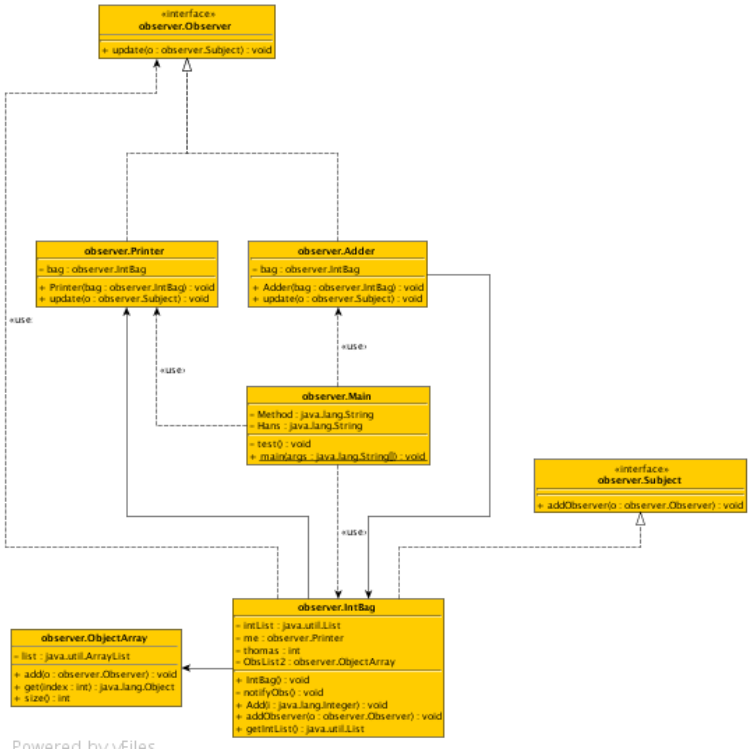

Figure A.9: The resulting Class Diagram

A.5 Uninstall

For the uninstall process go inside the *plugIns* folder in the root directory of the VizzAnalyzer tool, and manually remove the four files wich name start by *UMLConverter* and also remove the *yeduml* folder.

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/